Lab1: 简单集群搭建

1. 概况

(((截图太多了因为全是配置过程 我下个报告尽可能改💦💦💦

实验环境：`macOS 26.5.1, Debian GNU Linux`

使用到的Agent：`GitHub Copilot/Codex`

模型：`ChatGPT-5.x & GLM-5.2 & deepseek v4-pro`

通过此次环境配置和小集群的搭建，我意识到了AI的日益强大，也意识到自己亟需掌握更多Linux环境的知识和经验。文档的信息量不小，我需要慢慢消化。

于`2026.7.22`更新

2. 任务一：从源码构建 OpenMPI、BLAS 和 HPL

2.1 OpenMPI的构建

- 下载并`scp`压缩包

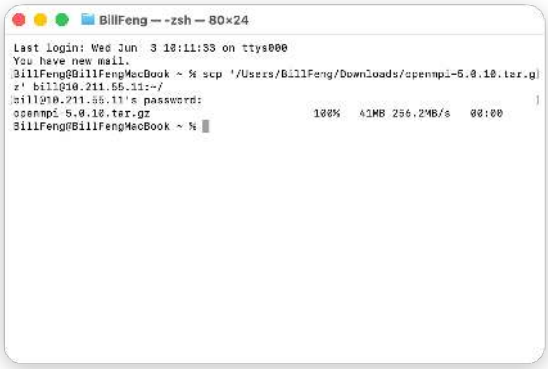

Image 1

- 解压并安装

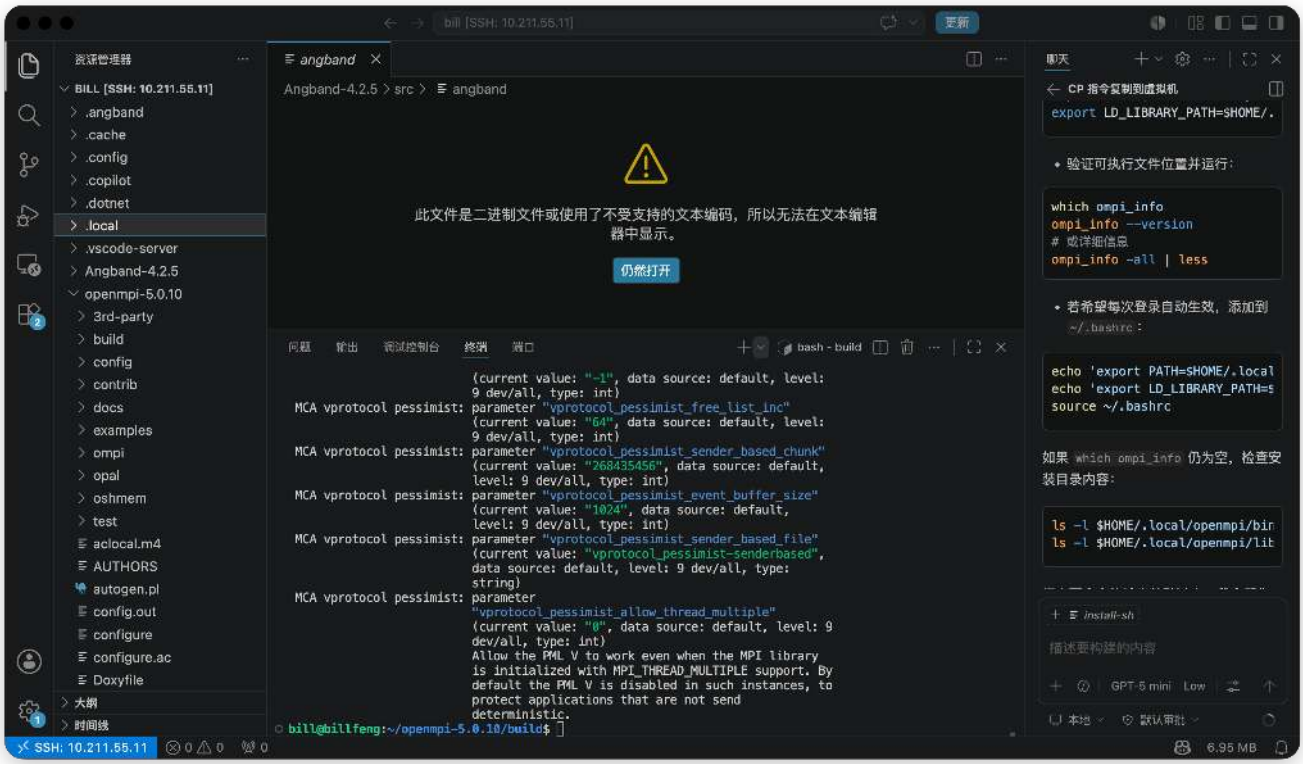

Image 2

2.2 BLAS&CBLAS

Image 3

- 修改编译器配置：

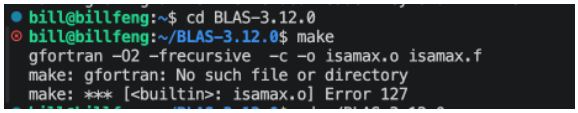

Image 4

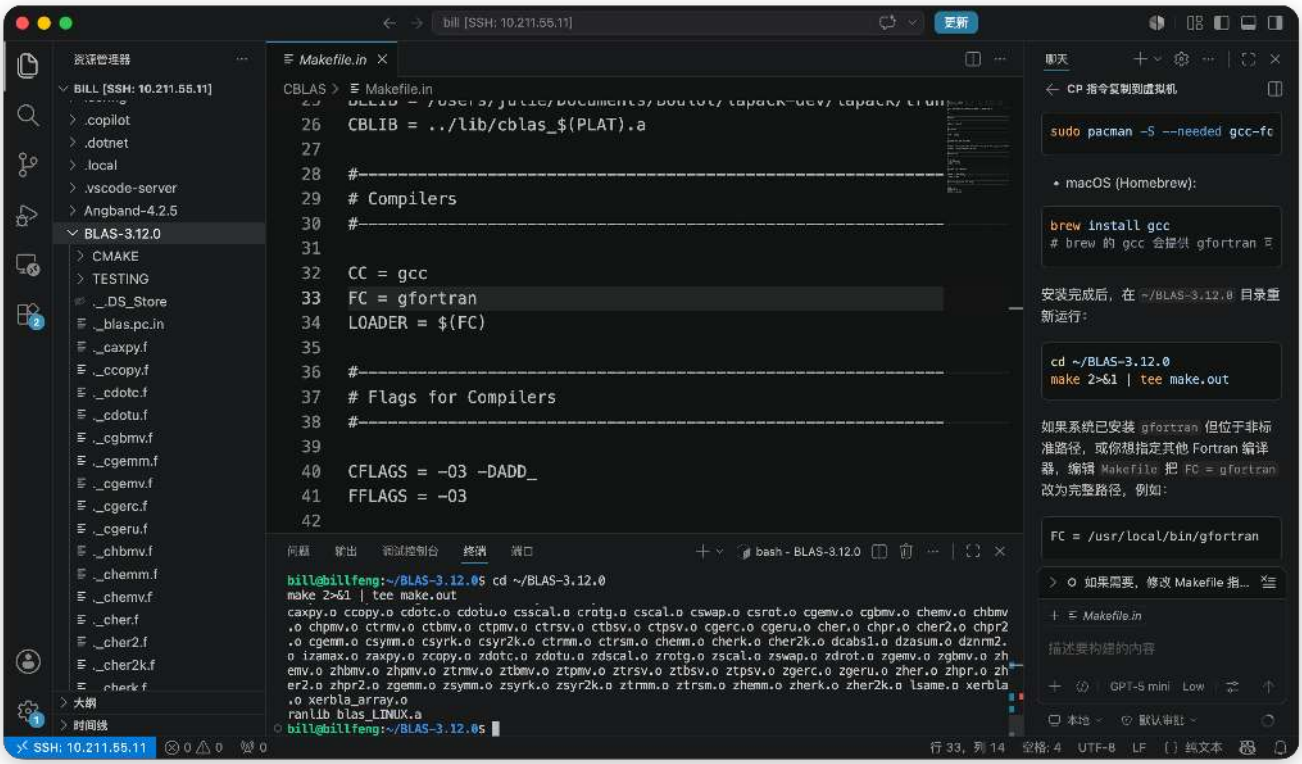

Image 5

- 遇到了文档中的问题，修改 `TOPdir` 路径：

```
bill@billfeng:~$ wget "https://netlib.org/benchmark/hpl/hpl-2.3.tar.gz"
tar xvf hpl-2.3.tar.gz
cd hpl-2.3
cp setup/Make.Linux_PII_FBLAS . …
make[1]: Entering directory '/home/bill/hpl-2.3'
( cd src/auxil/Linux_PII_FBLAS;         make )
make[2]: Entering directory '/home/bill/hpl-2.3/src/auxil/Linux_PII_FBLAS'
Makefile:47: Make.inc: No such file or directory
make[2]: *** No rule to make target 'Make.inc'.  Stop.
make[2]: Leaving directory '/home/bill/hpl-2.3/src/auxil/Linux_PII_FBLAS'
make[1]: *** [Make.top:54: build_src] Error 2
```

Image 6

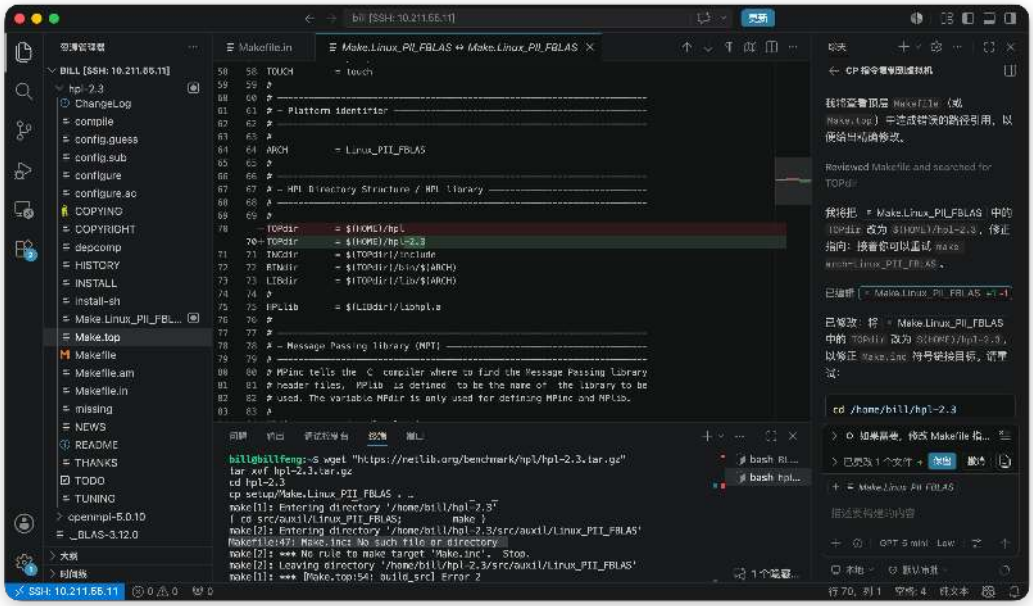

Image 7

2.3 HPL

下面是在 `Make.Linux_PII_FBLAS` 中修改或设定的参数：

- `TOPdir` : `$(HOME)/hpl-2.3`
- `MPdir` : `` `/home/bill/.local/openmpi ``
- `MPinc` : `` `-I$(MPdir)/include ``
- `MPlib` : `$(MPdir)/lib/libmpi.so`
- `LAdir` : `$(HOME)/BLAS-3.12.0`
- `LAlib` : `$(LAdir)/blas_LINUX.a $(LAdir)/cblas_LINUX.a`
- `LINKER` : `/usr/bin/gfortran`

成功生成可执行文件在 `bin/Linux_PII_FBLAS` 目录下：

Image 8

3. 任务二：配置 NFS 共享目录

- 这是四个节点映射的IP地址：

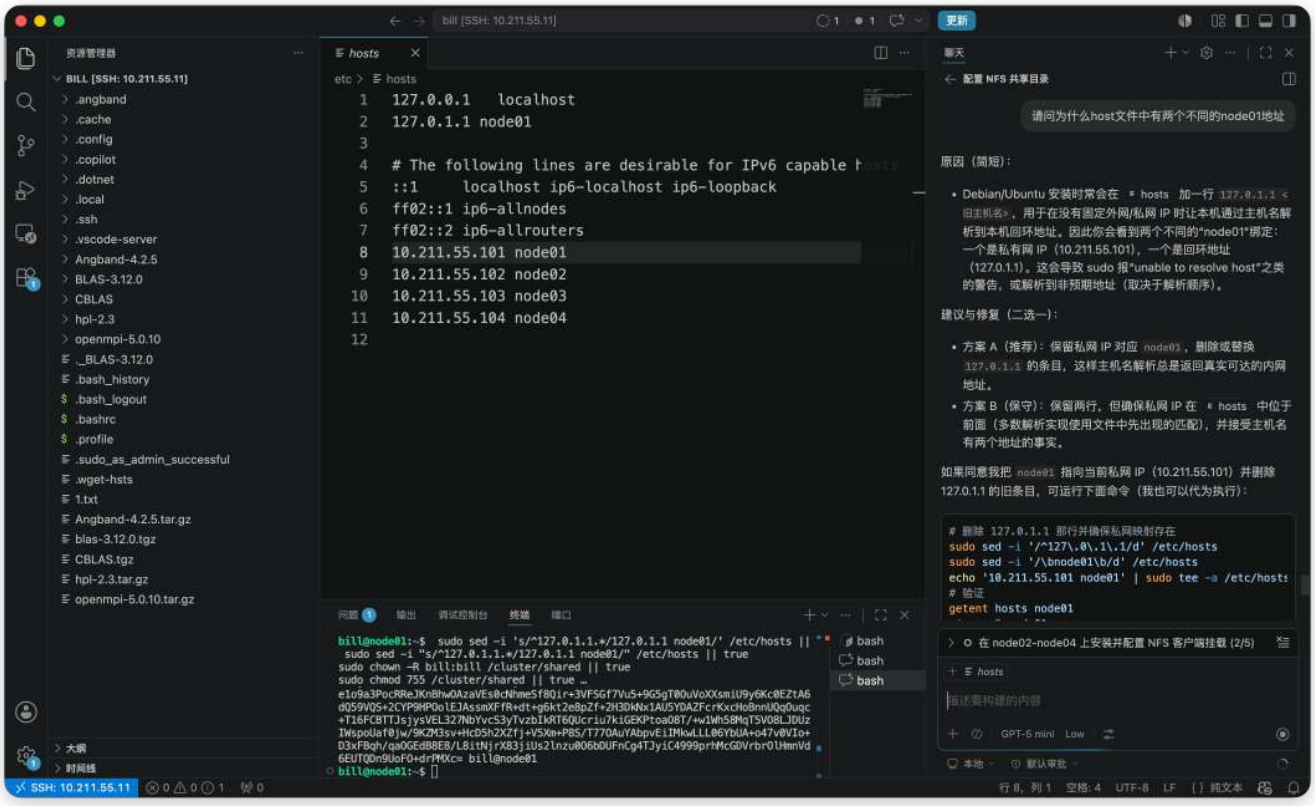

Image 9

- 我的思路是先配置 `node02`，然后将配置好的节点拷贝两份。下面三张图是我在 `node02` 中操作的记录。

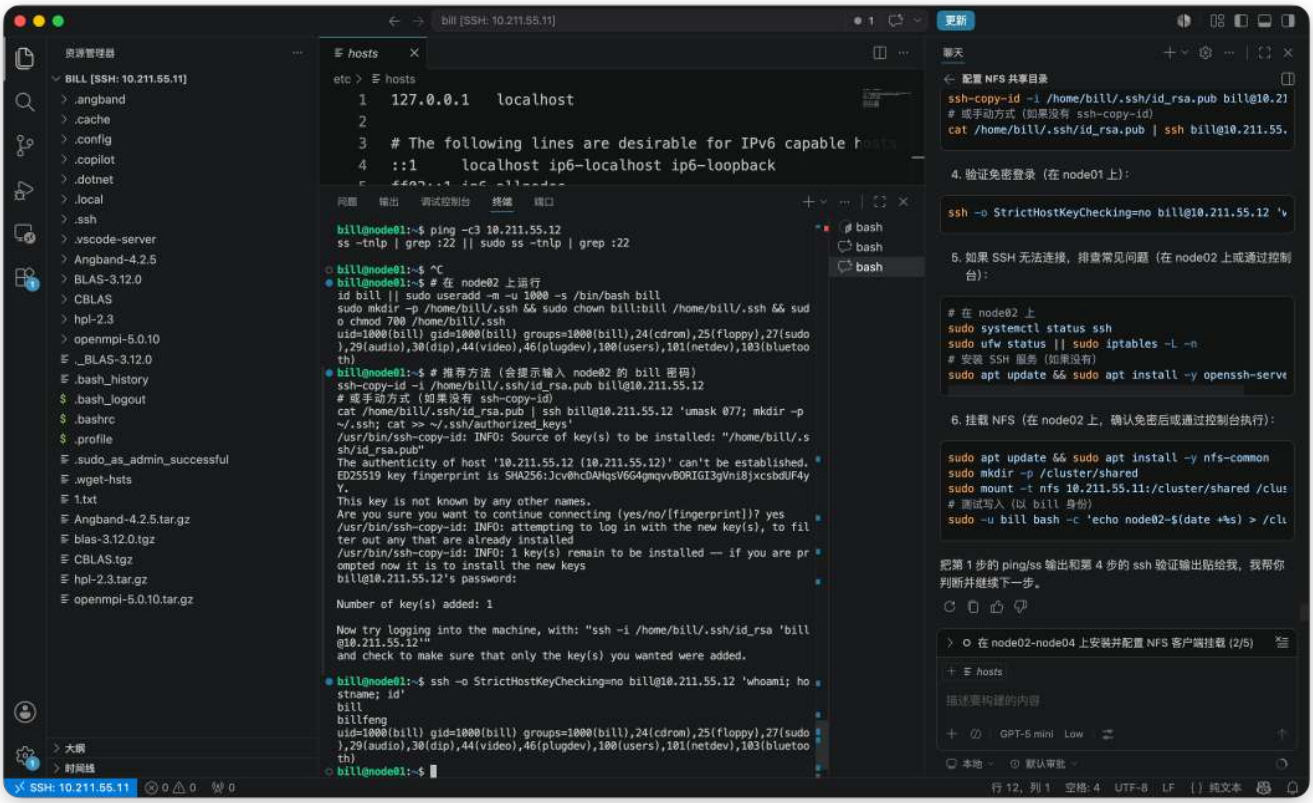

Image 10

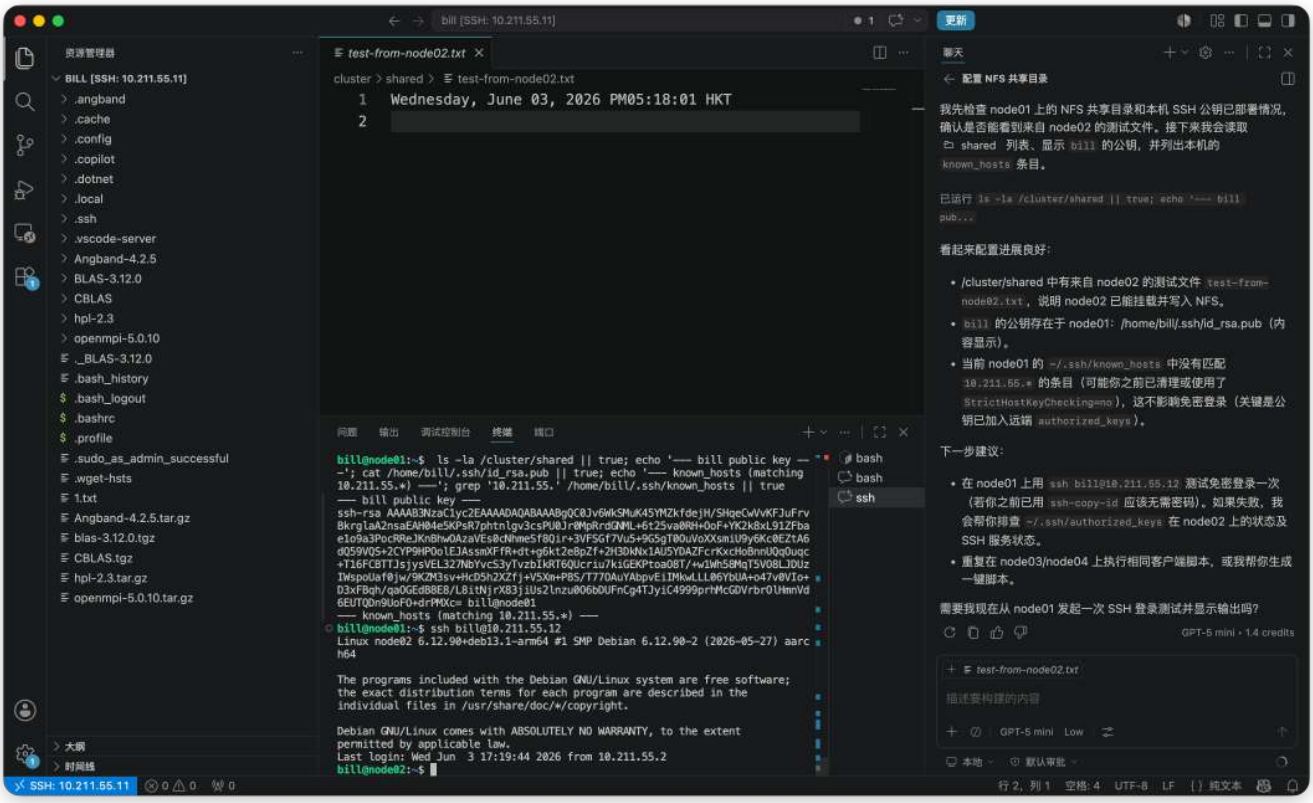

Image 11

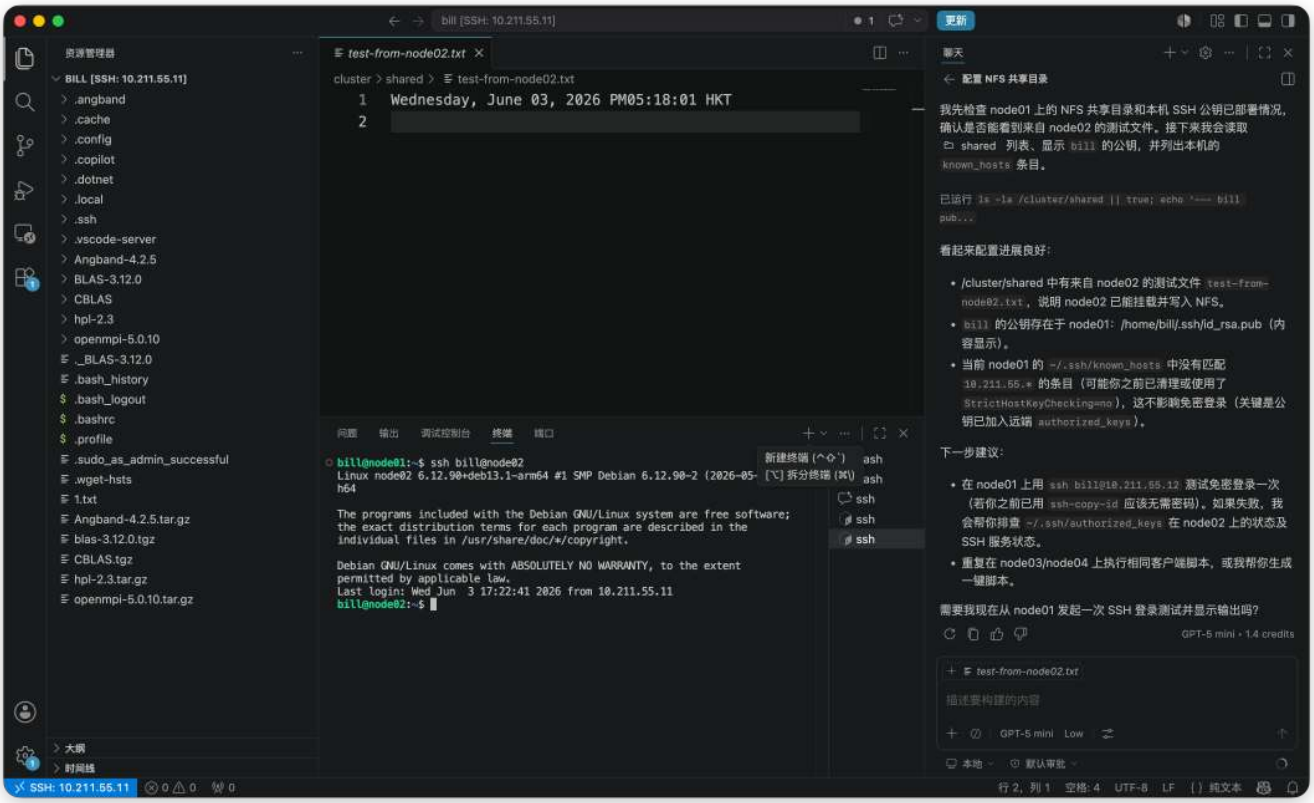

Image 12

- 将密钥生成算法修改为更推荐的`ed25519`，并在此处清理了多余的密钥。
- 在此处我还保持着对AI的掌控，尽量不用`Agent`而是用`Ask`模式，亲自输入或者粘贴命令，了解每一步都在做什么。到任务三点时候就有点“破戒”了。

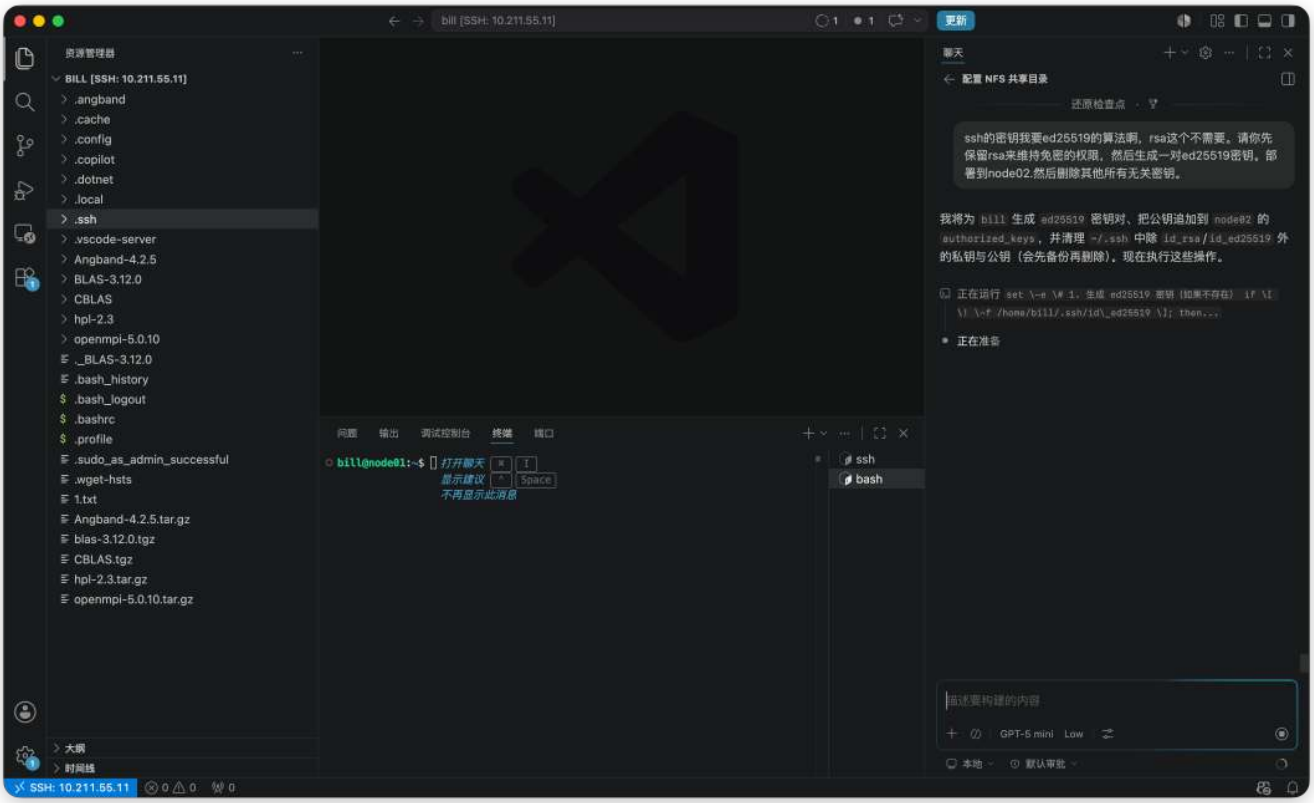

Image 13

- 见图片批注

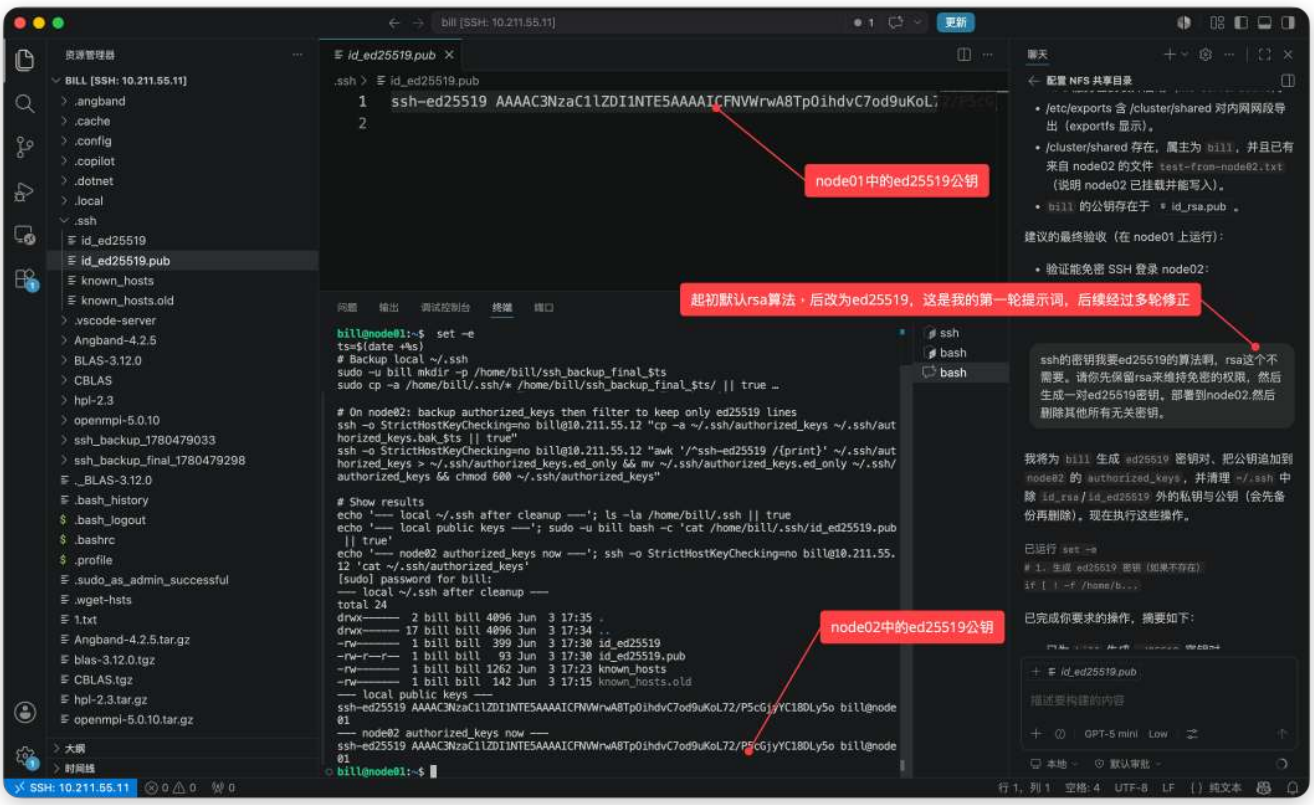

Image 14

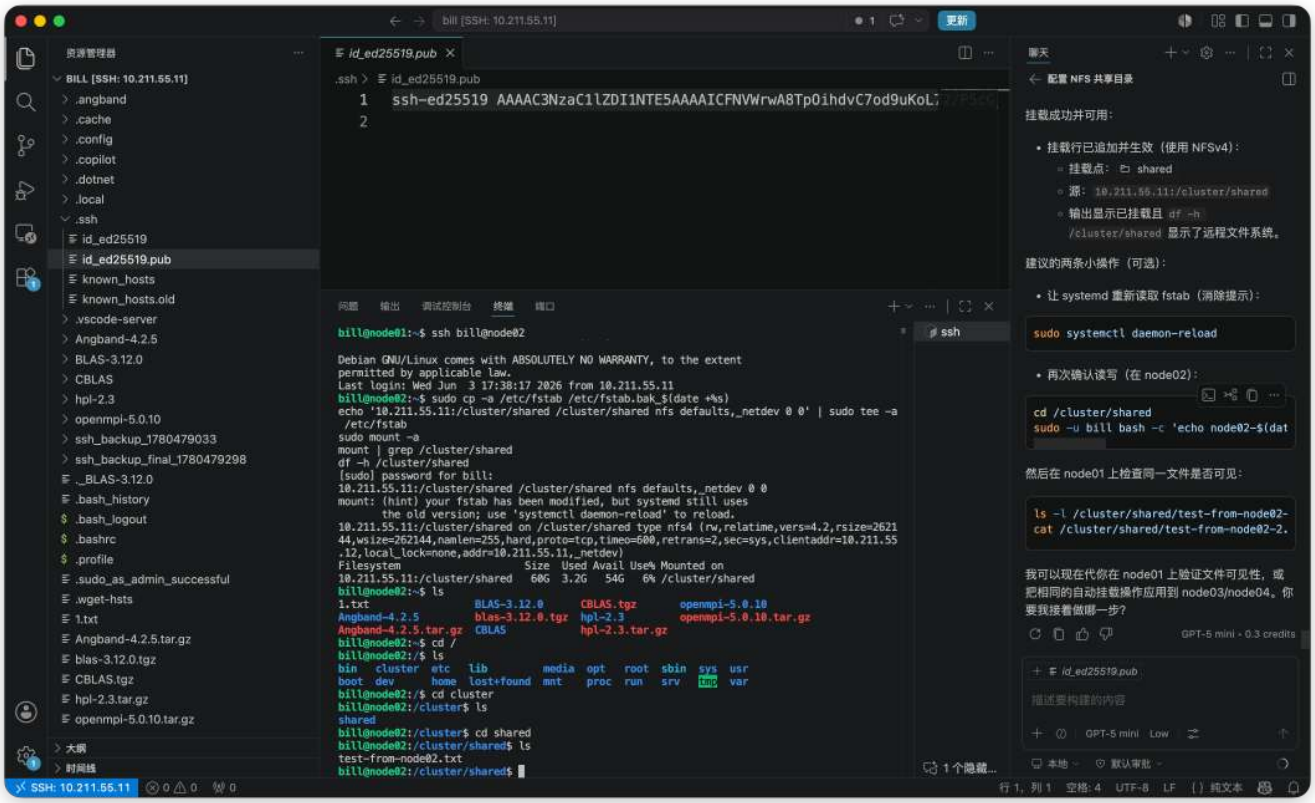

Image 15

- 拷贝 `node03 node04`，配置其地址和主机名。
- 4节点读写测试成功：

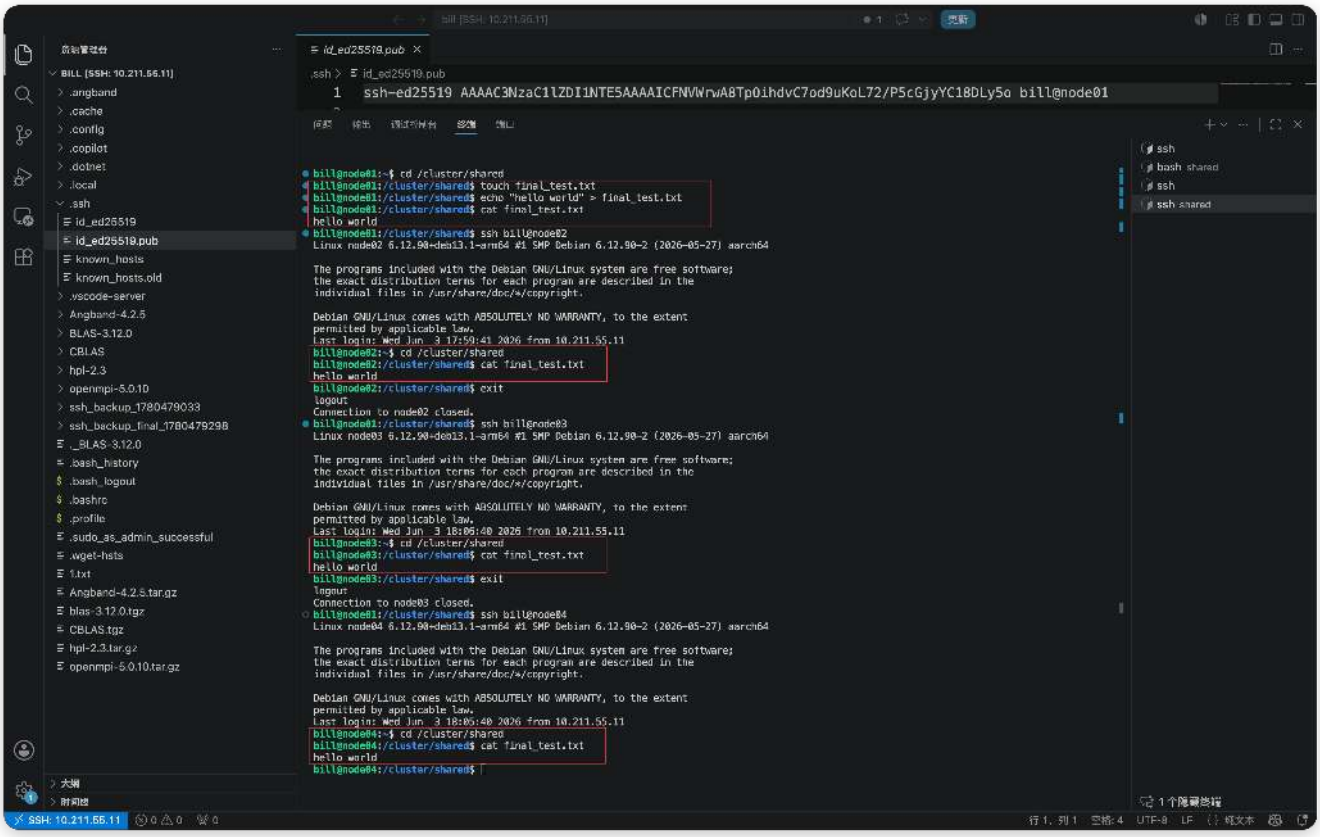

Image 16

- 读写文件时只和 `node01` 通信， `node02` 不通过 `node03` 读文件。

```
node02 ─┐
node03 ─┼─ node01(NFS 服务端)
node04 ─┘
//有点累，吃饭去了，明天继续。
```

4. 任务三：配置 Slurm 作业调度系统

- 集群搭建后 `srun` 单节点正常，但多节点作业报错。
- 我开始意识到自己的能力比较局限，借用了 `Agent` 来排查问题，发现是因为**UID 不一致导致 Slurm 跨节点通信失败**

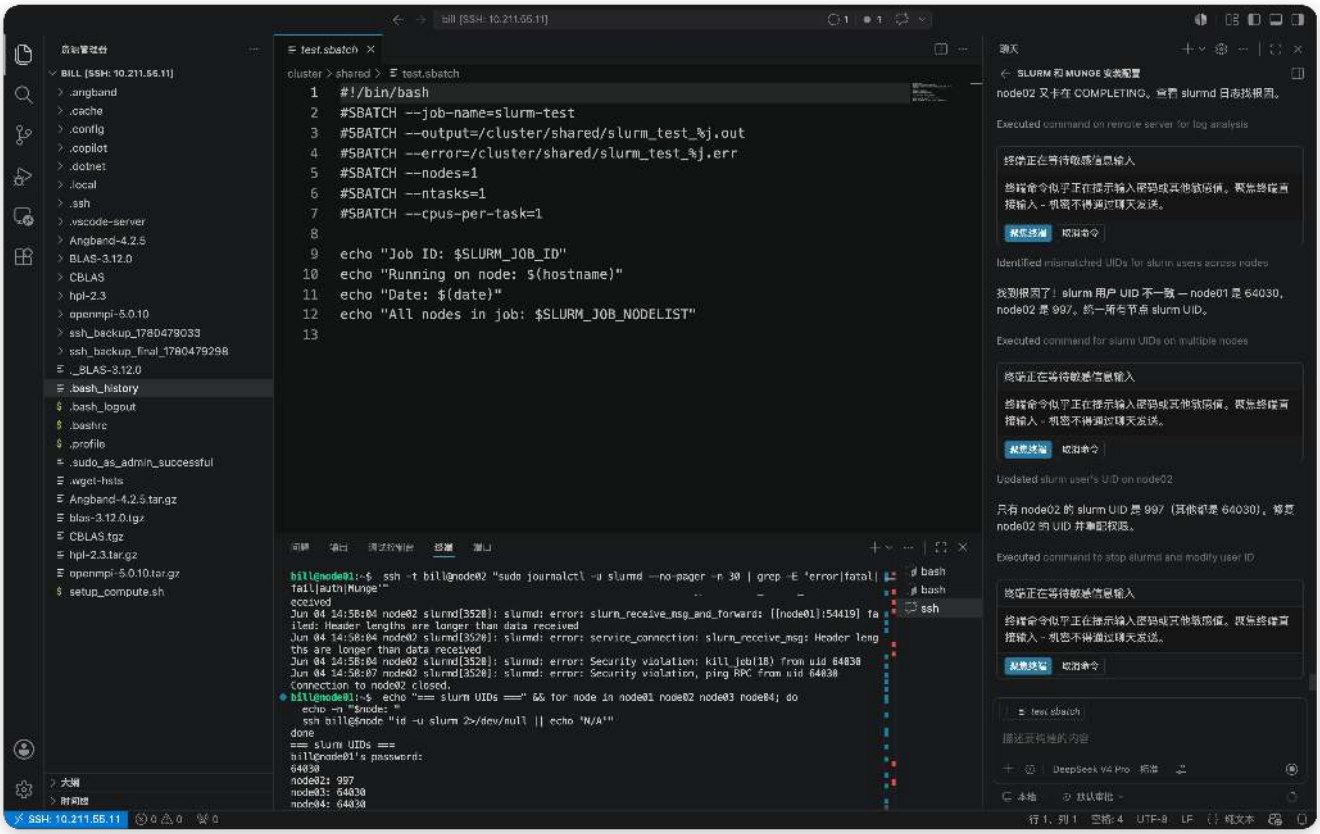

Image 17

修改 uid 一致后成功通信

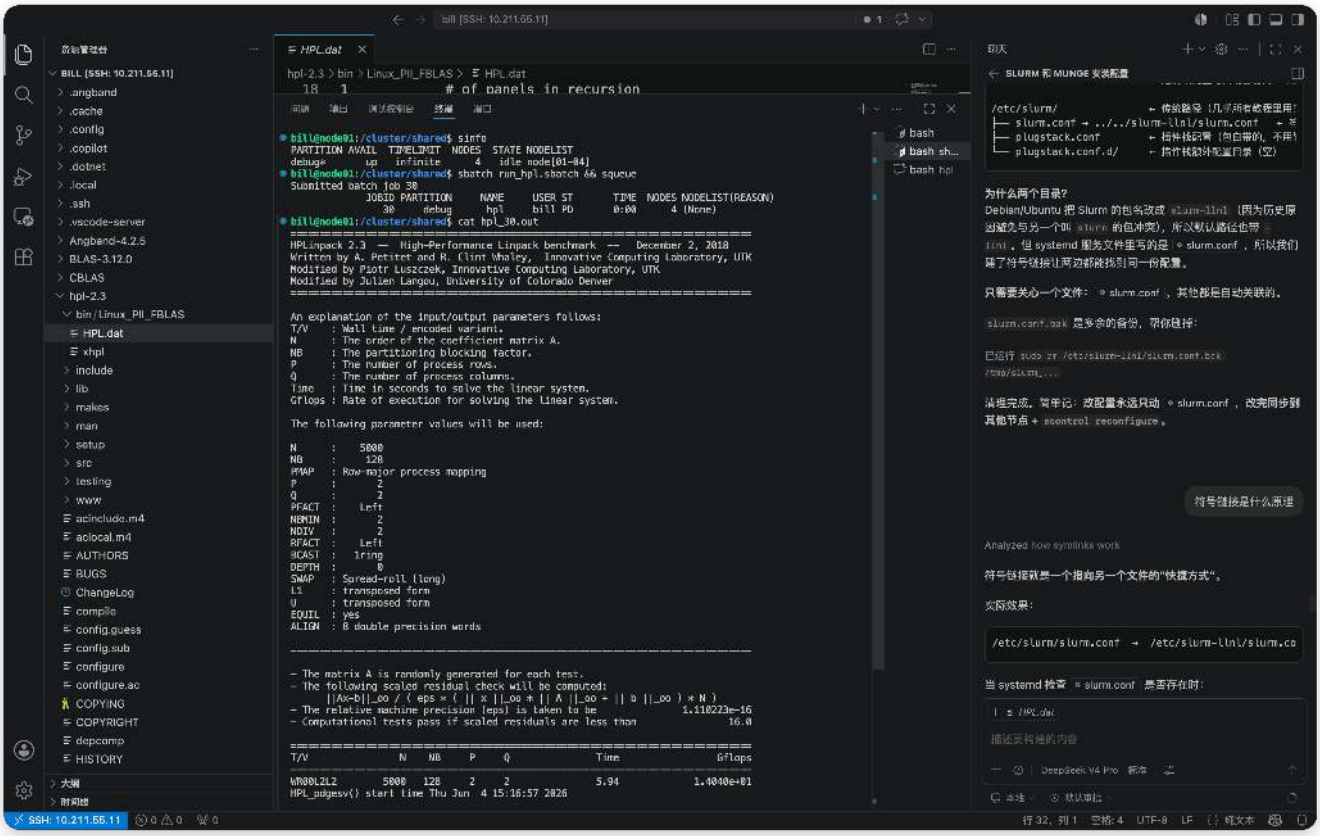

Image 18

5. 任务四：使用 Slurm 提交 HPL

- 关于 `HPL.dat`：

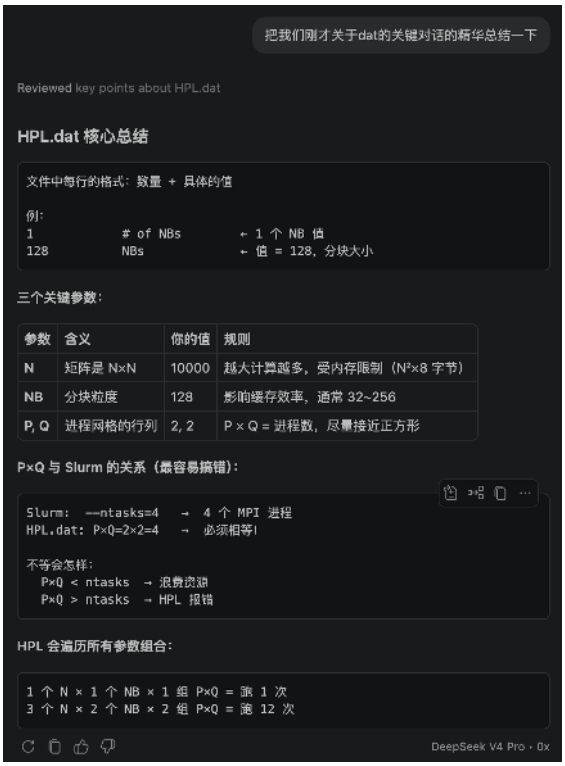

Image 19

- 根据文档推荐，去掉 `node01` 的计算节点属性，此时 `node01` 是登陆和控制节点， `node02-04` 是计算节点。

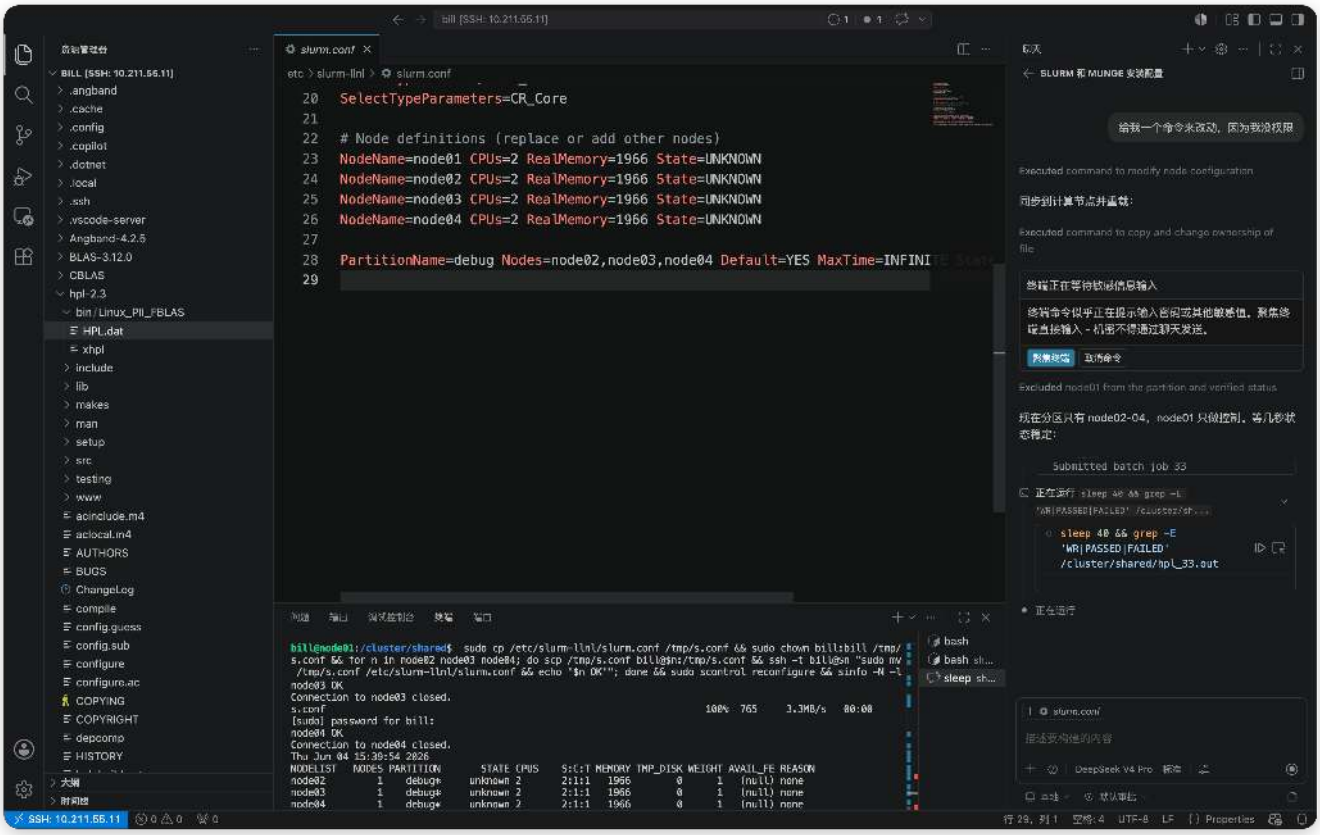

Image 20

- 用于提交HPL的脚本 `run_hpl.sbatch`

```
bill@node01:/cluster/shared$ ls
hpl  hpl_33.err  hpl_33.out  run_hpl.sbatch
bill@node01:/cluster/shared$ cat run_hpl.sbatch
#!/bin/bash
#SBATCH --job-name=hpl
#SBATCH --output=/cluster/shared/hpl_%j.out
#SBATCH --error=/cluster/shared/hpl_%j.err
#SBATCH --nodes=3
#SBATCH --ntasks=3
#SBATCH --cpus-per-task=1
#SBATCH --partition=debug

export PATH=/home/bill/.local/openmpi/bin:$PATH
export LD_LIBRARY_PATH=/home/bill/.local/openmpi/lib:$LD_LIBRARY_PATH

cd /cluster/shared/hpl
mpirun --oversubscribe ./xhpl
```

Image 21

- 见图片批注。

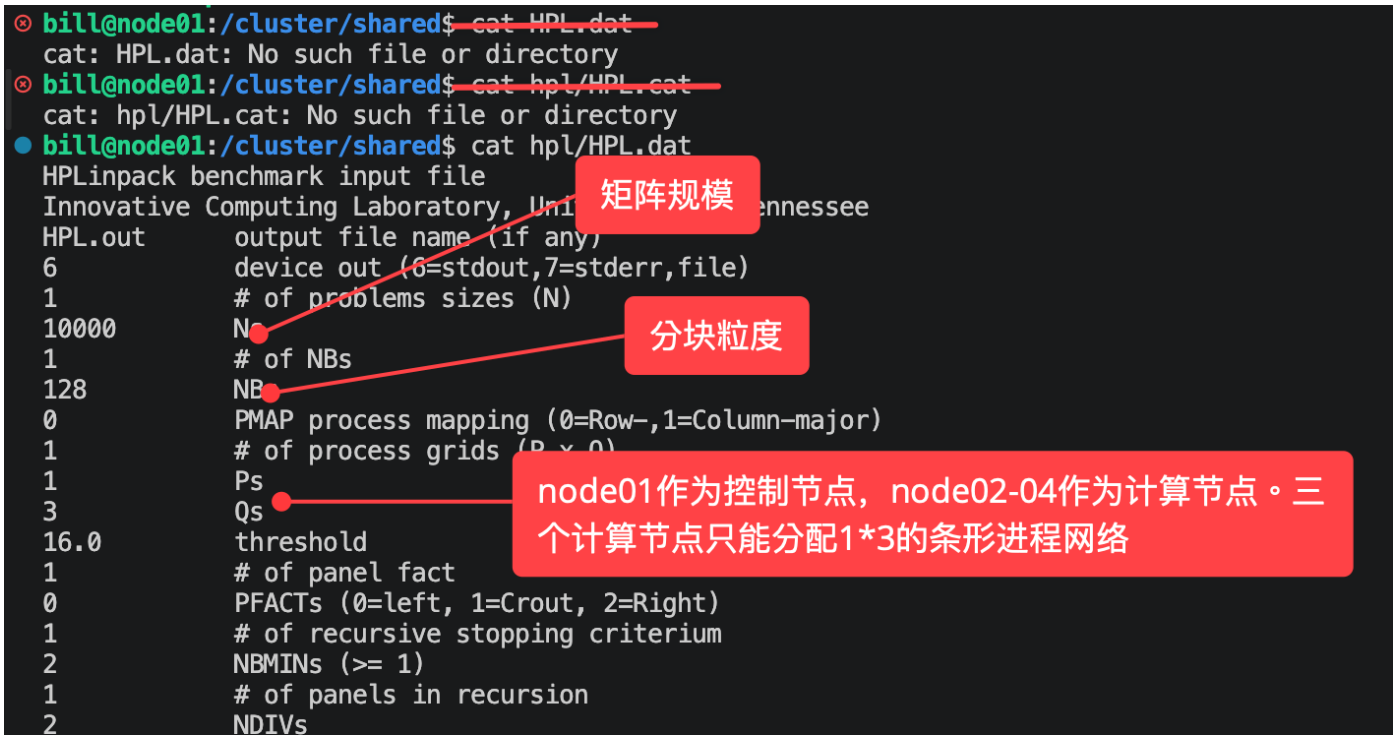

Image 22

- 随后通过任务管理器查看CPU资源情况，发现资源在 `node02` 上堆满，而其他节点几乎没有任务。

Image 23

- 来自AI的解释：`$SLURM_JOB_NODELIST 返回压缩格式 node[02-04]，OpenMPI不认得。展开它。`
- 修改后性能有所提升，见下图：
  - 上面框内部分是 `node02` 单独计算

- 下面框内部分是 `node02-04` 协同计算

```
- The relative machine precision (eps) is taken to be           1.110223e-16
- Computational tests pass if scaled residuals are less than                16.0

================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR00L2L2       10000   128     1     3              50.97              1.3083e+01
HPL_pdgesv() start time Thu Jun  4 15:49:16 2026

HPL_pdgesv() end time   Thu Jun  4 15:50:07 2026

--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=   4.51323610e-03 ...... PASSED
================================================================================

--------------------------------------------------------------------------------

- The matrix A is randomly generated for each test.
- The following scaled residual check will be computed:
      ||Ax-b||_oo / ( eps * ( || x ||_oo * || A ||_oo + || b ||_oo ) * N )
- The relative machine precision (eps) is taken to be           1.110223e-16
- Computational tests pass if scaled residuals are less than                16.0

================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR00L2L2       10000   128     1     3              39.83              1.6741e+01
HPL_pdgesv() start time Thu Jun  4 15:53:00 2026
```

Image 24

- 任务管理器的资源分配说明进程调度是有效的！

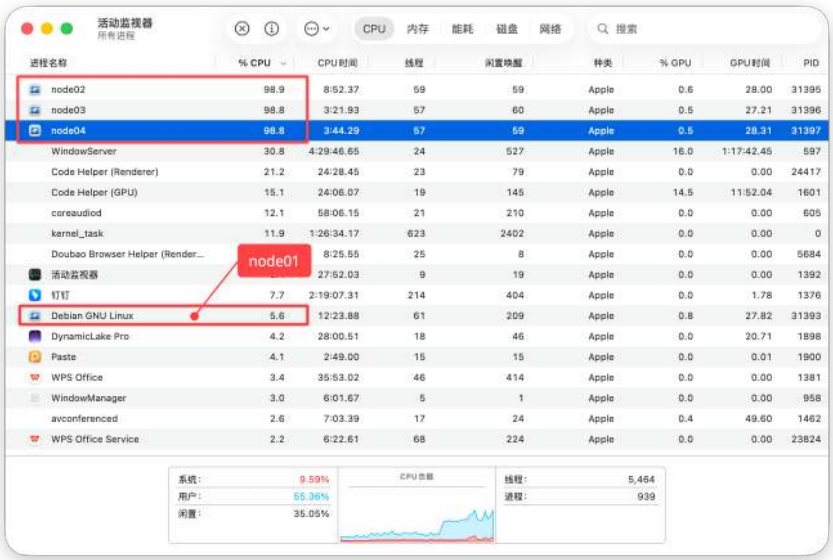

Image 25

对于调优等任务将在以后更新。

2026.7.22 更新

由于前文的 `slurm.conf` 文件里指出每个节点是2核心的（`CPUs=2`），所以进程网络可以调整为 `2×3` 或 `3×2`。

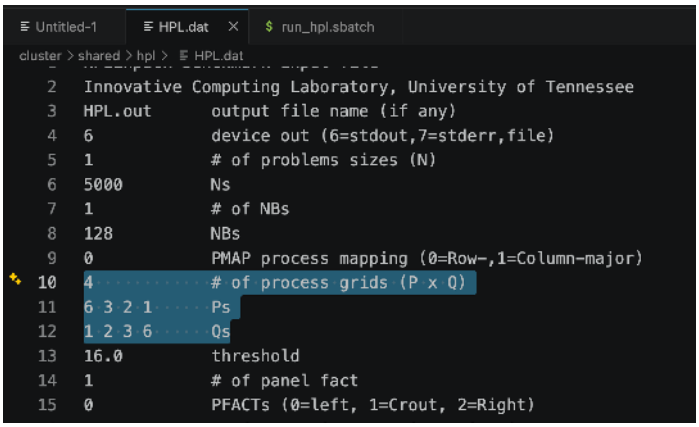

Image 26

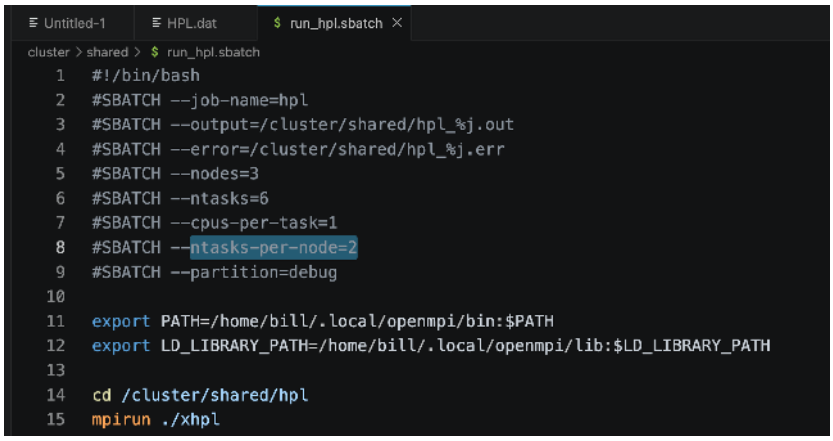

Image 27

以下是部分AI的对话。

0.0.1 那之前是两个cpu应对一个task吗，就是1*3的时候？还是说有性能浪费？

对，之前 `--ntasks=3`（每节点1进程）时，每个节点**只用了 1 个核，另 1 个核空闲**。对比一下：

配置	每节点进程	CPU使用率	性能
1×3 (旧)	1 进程	**只用 1/2 核**	~13 Gflops
2×3 (新)	2 进程	**用满 2/2 核**	~22-24 Gflops

Table 1

性能几乎翻倍，因为之前确实浪费了一个核。

不过现在 `--ntasks=6` + `--cpus-per-task=1` 就是**完美 1:1 映射**——每个进程独占一个 CPU 核心，没有浪费也没有超分，是最优配置。

0.0.2 是不是每次跑完之后要停止一段时间？不同的任务是先后关系吗？你说的“1. 顺序影响性能——第一个跑的配置往往更快”，这是为什么呢？

好问题！分析一下：

1. 不同配置之间是先后关系吗？

是的，`mpirun ./xhpl` 一次运行中，4 个 P×Q 配置是**顺序执行**的——跑完一个紧接着跑下一个，中间没有停顿。

2. 为什么第一个最快？

看你的数据趋势就很明显了：

测试顺序	P×Q	耗时	Gflops	
①	1×6	28.81s	23.15	🥇 最快
②	2×3	30.30s	22.00	↓
③	3×2	32.50s	20.52	↓
④	6×1	39.64s	16.82	🥉 最慢

性能**依次递减**，跟 P×Q 关系不大，关键是**越往后越慢**。

最可能的原因是 **CPU 过热降频**——这些小 ARM 设备散热有限，第一个测试时 CPU 还凉快，跑满频率；连续全速跑 2 分多钟后，温度上去就会**自动降频**，后面越来越慢。

0.1 重新跑一次，得到如下结果

为了验证是否是CPU降频的问题，我将四组数据倒序测试：

```
bill@node01:/cluster/shared/hpl$  cd /cluster/shared/hpl && sbatch /cluster/shared/run_hp
l.sbatch && sleep 30 && grep "WR00" /cluster/shared/hpl_51.out 2>/dev/null
Submitted batch job 51
WR00L2L2        5000   128     1     6             3.40             2.4534e+01
WR00L2L2        5000   128     2     3             3.57             2.3374e+01
WR00L2L2        5000   128     3     2             3.97             2.0975e+01
WR00L2L2        5000   128     6     1             4.88             1.7100e+01
bill@node01:/cluster/shared/hpl$  cd /cluster/shared/hpl && sbatch /cluster/shared/run_hp
l.sbatch && sleep 30 && grep "WR00" /cluster/shared/hpl_52.out 2>/dev/null
Submitted batch job 52
WR00L2L2        5000   128     6     1             5.07             1.6431e+01
WR00L2L2        5000   128     3     2             4.03             2.0670e+01
WR00L2L2        5000   128     2     3             3.70             2.2521e+01
WR00L2L2        5000   128     1     6             3.60             2.3189e+01
```

Image 28

由此可见，排名不受顺序影响——1×6 > 2×3 > 3×2 > 6×1。

再次询问AI，“在这个小 ARM 集群上，内存访问的连续性比通信拓扑更重要。对于较大问题，2×3 的优势才会显现出来。”

更改N的值：

```
bill@node01:/cluster/shared/hpl$ grep "WR00" /cluster/shared/hpl_53.out 2>/dev/null
WR00L2L2        8000   128     6     1             15.10            2.2618e+01
WR00L2L2        8000   128     3     2             14.69            2.3235e+01
WR00L2L2        8000   128     2     3             14.86            2.2981e+01
WR00L2L2        8000   128     1     6             16.70            2.0447e+01
```

Image 29

N=8000 时，3×2 和 2×3 表现最好（约 23 Gflops），方阵网格开始显现优势，1×6 的列连续优势在较大矩阵下已消失。

取P=3, Q=2，调整分块大小 NB 以优化计算效率：

```
bill@node01:/cluster/shared/hpl$  sleep 120 && grep "WR00" /cluster/shared/hpl_55.out 2>/dev/null
WR00L2L2        8000    32      3       2               13.87           2.4623e+01
WR00L2L2        8000    64      3       2               14.08           2.4244e+01
WR00L2L2        8000    96      3       2               14.80           2.3069e+01
WR00L2L2        8000    128     3       2               15.42           2.2142e+01
WR00L2L2        8000    160     3       2               15.89           2.1489e+01
WR00L2L2        8000    192     3       2               17.10           1.9965e+01
WR00L2L2        8000    256     3       2               19.67           1.7355e+01
```

Image 30

1. Bonus：更换数学库

这里选择了 **OpenBLAS**，因为其对于 arm 架构有比较好的支持，且安装和获取相对容易。

```
cd /home/bill/OpenBLAS-0.3.28
make clean
make -j1 CC=gcc FC=gfortran TARGET=ARMV8 USE_OPENMP=0 NO_LAPACK=0 DYNAMIC_ARCH=0
```

接着修改 hpl 的 make 配置的库路径：

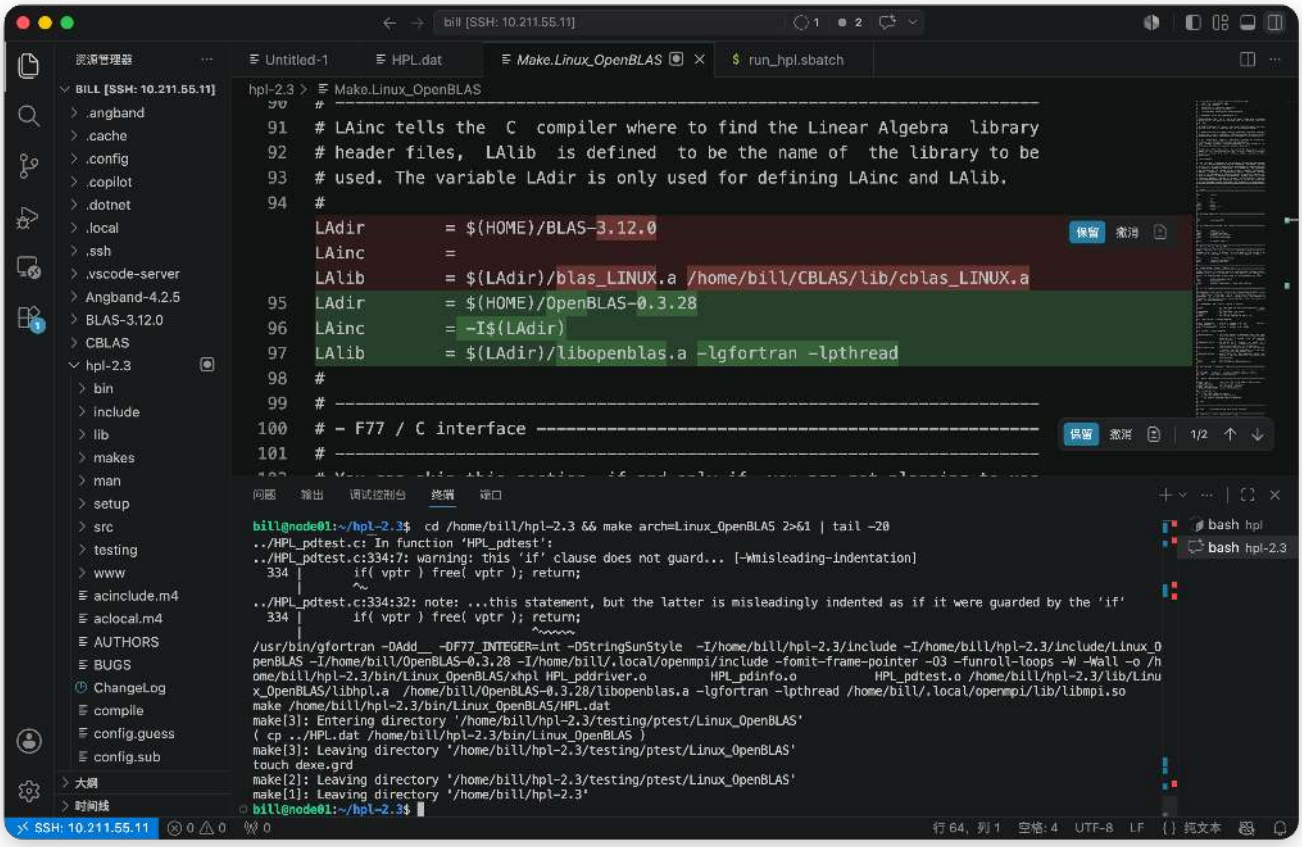

Image 31

接着创建新的 `xhpl_openblas` 文件和新的 `run_hpl_openblas.sbatch` 测试脚本。修改对应的库路径，然后开始测试，得出结果：

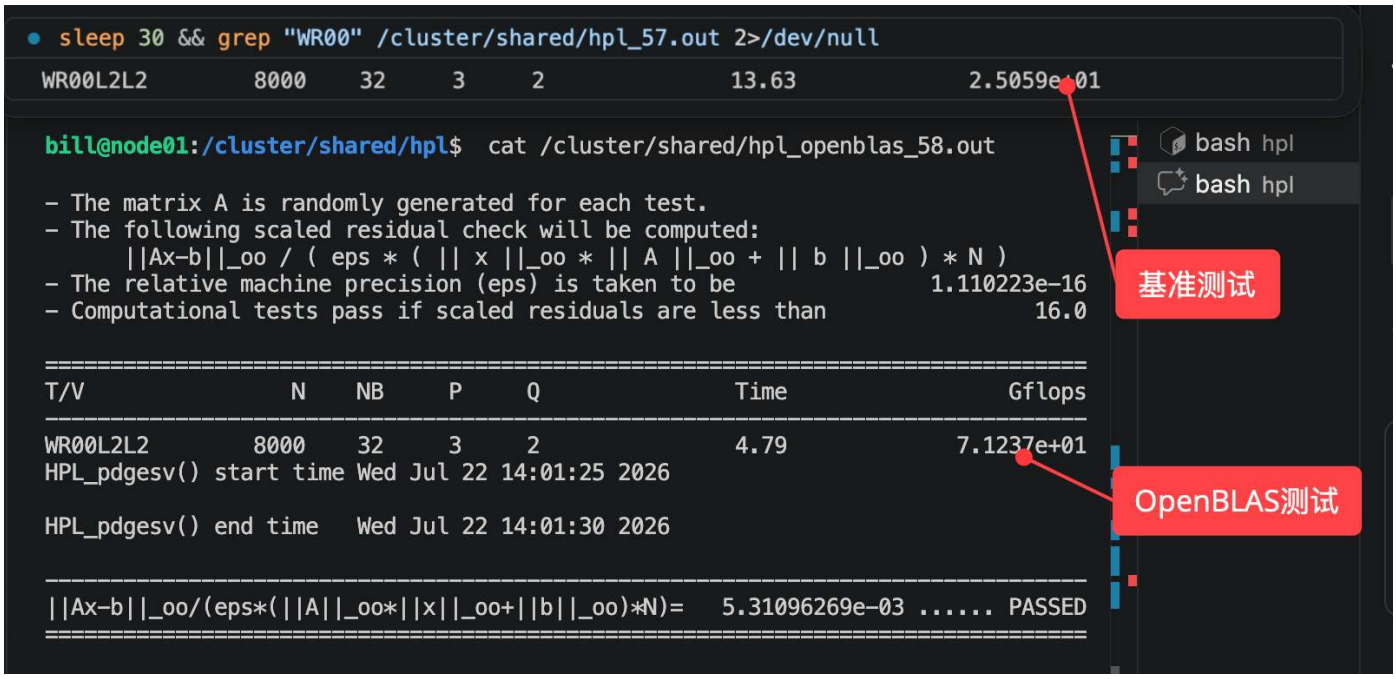

Image 32

```
## 为什么性能提升这么大（25 → 71 Gflops，+184%）？

唯一的变量是数学库。参考 BLAS 是纯 Fortran 写的朴素实现，没有架构优化。OpenBLAS 包含手写的 ARM NEON 汇编内核，利用了 SIMD 向量化、缓存分块、内存预取等技术，在 ARMv8 架构上差距自然很大。
```

2. Bonus：更换共享文件系统

这里选择了 **JuiceFS**，因为其部署轻、无需额外服务节点、提供 aarch64 静态二进制，适合在小规模 ARM 集群上快速搭建。

```
# 下载
wget https://github.com/juicedata/juicefs/releases/download/v1.2.2/juicefs-1.2.2-linux-arm64.tar.gz
tar -xzf juicefs-1.2.2-linux-arm64.tar.gz
sudo cp juicefs /usr/local/bin/

# 初始化（SQLite 元数据 + NFS 共享路径存储数据块）
juicefs format --storage file --bucket /cluster/shared/jfs_data sqlite3:///cluster/shared/jfs.db finalfs

# node01 和 node02 分别挂载
juicefs mount -d sqlite3:///cluster/shared/jfs.db /mnt/jfs
```

输出了`Do not have write permission on /mnt/jfs [mount_unix.go:472]`

权限问题，node02 上 jfs 属于 root。在 node02 终端执行：

```
sudo chown bill:bill /mnt/jfs # 释放sudo权限
juicefs mount -d sqlite3:///cluster/shared/jfs.db /mnt/jfs
# 挂载，db是存储文件元数据的数据库后缀，/mnt/jfs是挂载目录
```

挂载成功，但是在验证跨节点通信时，却得到`mnt/jfs/cross.txt: Input/output error`的提示。

所以我只好又去询问万能的ai：

> 因为 SQLite 元数据引擎是单文件锁定模式。node01 挂载时持有 SQLite 数据库的写锁，node02 再挂载时只能以只读模式工作。当你尝试在 node02 上 `cat` 一个文件时，JuiceFS 需要从元数据库读取文件的块位置信息，但此时 SQLite 锁被 node01 持有的写操作占用，node02 无法正常访问元数据，导致 IO 错误。
>
> 简单说：**SQLite 同一时刻只允许一个客户端写入，JuiceFS 的多挂载依赖元数据引擎的并发能力。** 要真正支持多节点并发，需要用 Redis、MySQL 等支持并发的元数据引擎替代 SQLite。

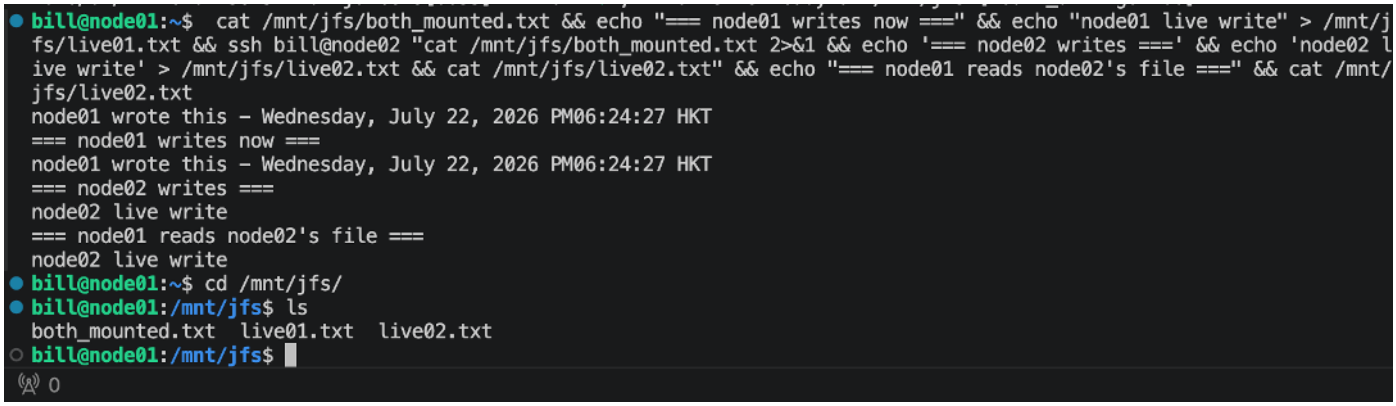

Image 33

3. Bonus：k3s

因为 k3s 需要一些额外的内存，这个时候我去更改内存大小的配置。结果 ssh 突然用不了了。具体情况是虚拟机之间能ping通；主机和任一虚拟机节点均无法ping通。我一度怀疑是我改内存配置过于着急而强制关机产生了这些问题（是的，关机还要sudo密码和一行命令，懒得打了直接中断运行），前前后后折腾了一个小时（我已急哭）。最后发现是 rvpn 将这部分网段的流量放进管道了。。。

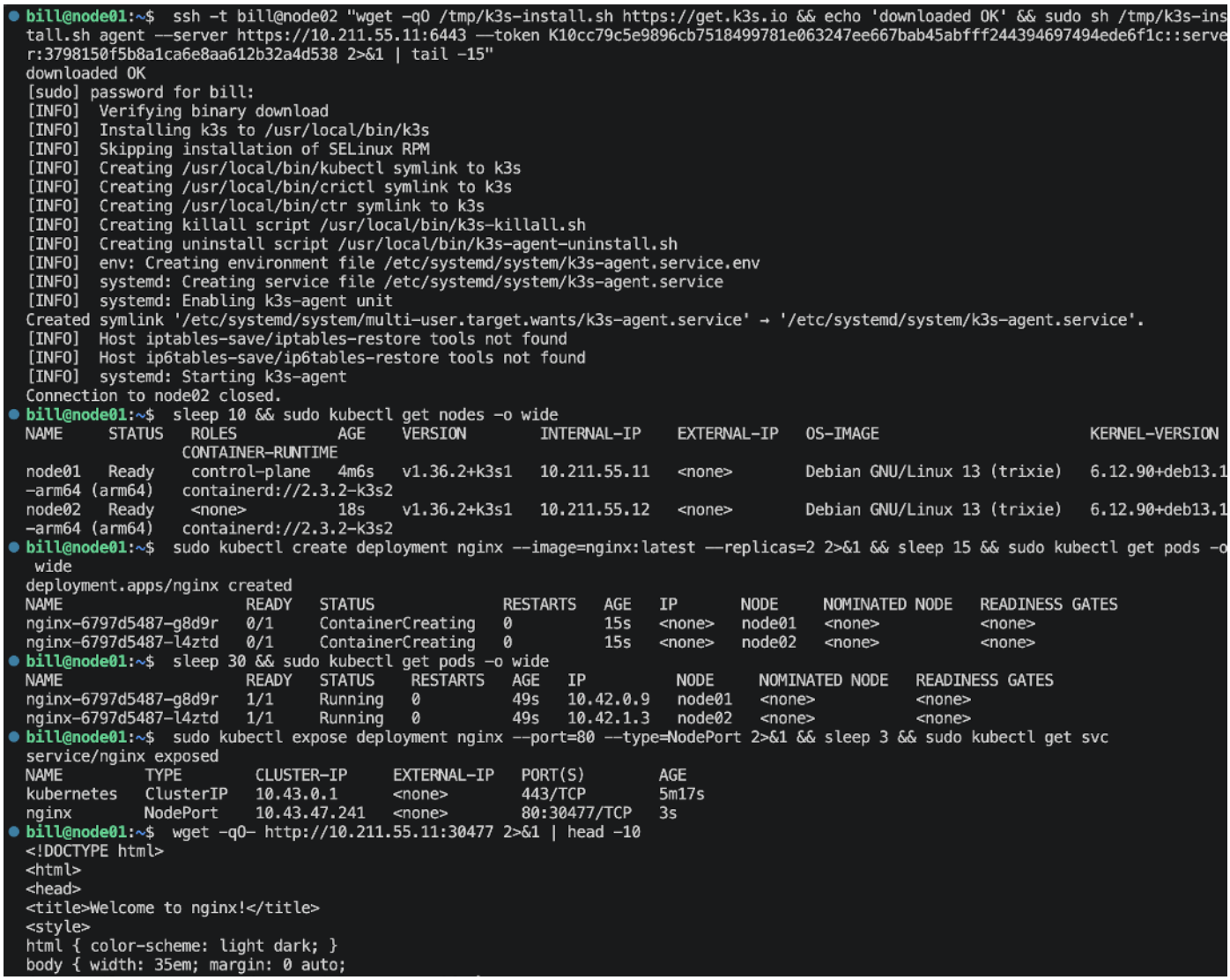

Image 34

agent token 是 k3s server 生成的一个认证令牌，agent 节点用它来证明自己有权加入集群。

获取方式是在 node01（server 节点）上读取：

```
sudo cat /var/lib/rancher/k3s/server/node-token
```

Q: NodePort 和 ClusterIP 有什么区别？为什么集群内部的服务能被外部访问？

以"在集群里跑一个 Web 面板让用户通过浏览器提交作业"为例：

ClusterIP：只在集群内部可访问。比如集群里的 Prometheus 监控采集 JupyterHub 的指标，不需要外部访问，用 ClusterIP 就行。

NodePort：在每个节点的真实网卡上开放一个端口（30000-32767），外部可以通过节点 IP + 该端口访问。比如用户要通过浏览器访问 JupyterHub，就需要 NodePort 把服务暴露出去。

原理是 kube-proxy 在每个节点上写 iptables 规则，做了一层 DNAT（目标地址转换）：

```
用户浏览器 → http://10.211.55.11:30477（真实节点 IP + 真实端口）
                          |
                    iptables DNAT
                          |
                    10.42.0.9:80（Pod 虚拟 IP，集群内部）
```

Pod 的 IP 是容器网络的虚拟地址，外部路由不到。但节点的 IP 是真实的，iptables 把外部能到达的真实地址映射到了集群内部的虚拟地址，这就是外部能访问集群服务的原因。

4. 进阶挑战：身份认证 / 目录服务部署

选了 LDAP。直接在 node01 上用 apt 装 slapd。

4.1 部署过程

4.1.1 安装

```
sudo apt install -y slapd ldap-utils
```

装完之后用 `dpkg-reconfigure slapd` 重新配置了一下：

- 域名：`hpclab.local`，对应 LDAP 的根节点就是 `dc=hpclab,dc=local`
- 管理员密码：`admin123`
- 组织名：HPC Lab
- 后端数据库：MDB

配完之后用 admin 密码连不上，报 `Invalid credentials (49)`。查了一下发现是 `olcRootPW` 没设对，用 `slappasswd` 生成哈希后通过 `ldapmodify` 改了一下才好。

4.1.2 创建测试用户和组

LDIF 文件（`test-user.ldif`）详见下方图片：

```
dn: ou=people,dc=hpclab,dc=local
objectClass: organizationalUnit
ou: people
...
member: uid=testuser,ou=people,dc=hpclab,dc=local
```

然后导入：

```
ldapadd -x -D "cn=admin,dc=hpclab,dc=local" -w admin123 -f test-user.ldif
```

创建了两个 OU（people 和 groups），一个用户 testuser，一个组 hpcusers，testuser 在 hpcusers 组里。

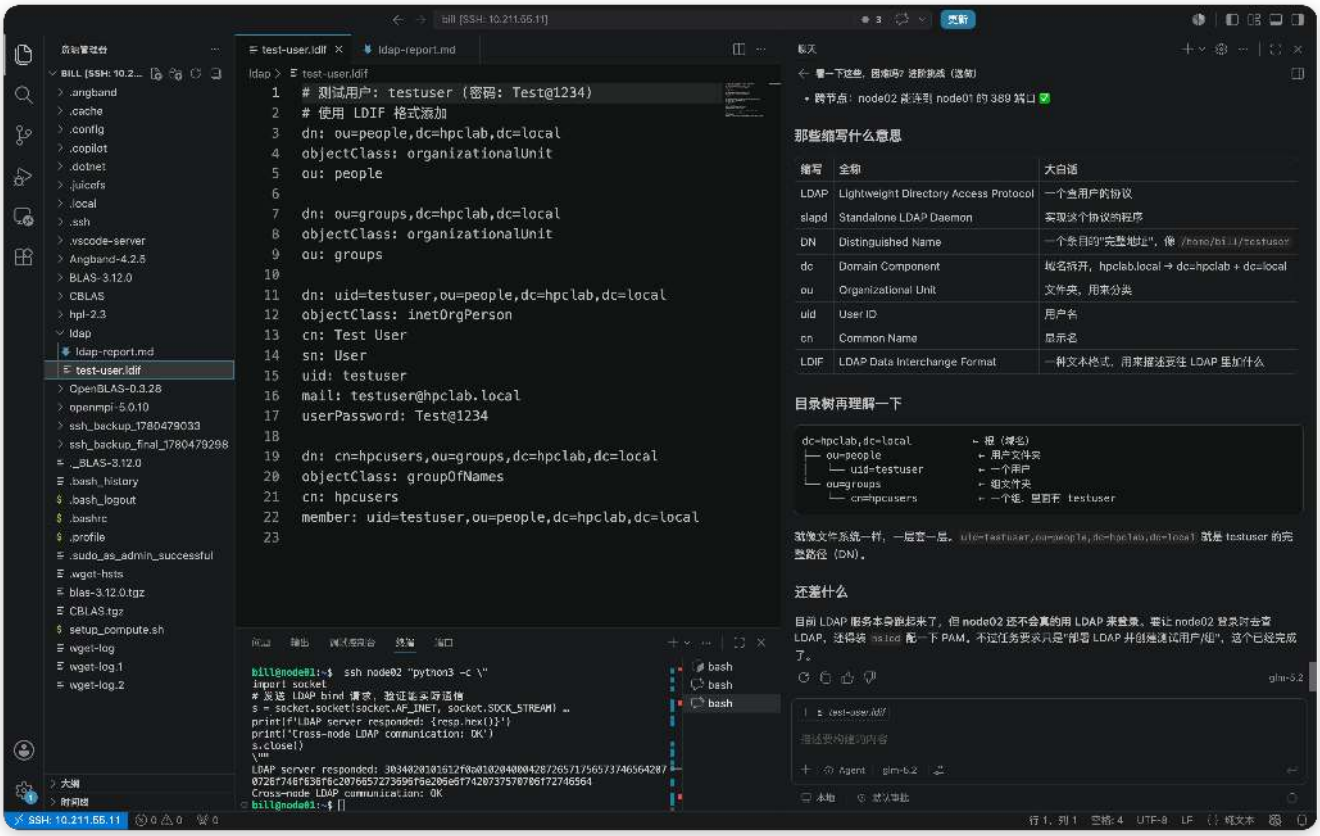

Image 35

4.2 目录结构

LDAP 的数据是树形的

```
dc=hpclab,dc=local
├── ou=people
│   └── uid=testuser
└── ou=groups
    └── cn=hpcusers (member: testuser)
```

dc 就是域名拆开（hpclab.local → dc=hpclab,dc=local），ou 类似文件夹，uid 是用户，cn 是组的名字。

4.3 验证

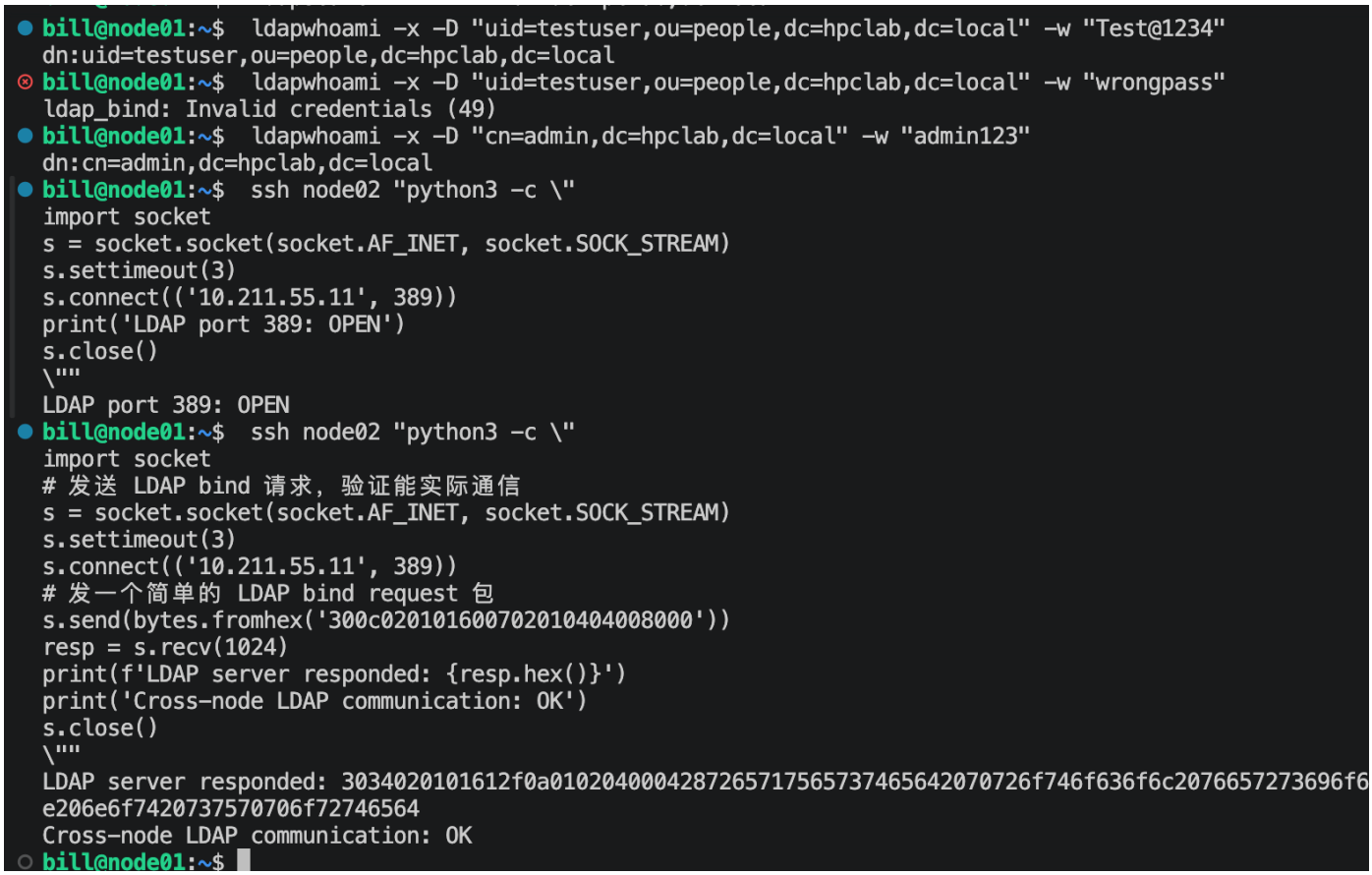

Image 36

4.3.1 查询

```
ldapsearch -x -LLL -b "dc=hpclab,dc=local"
```

能查到所有条目，包括 testuser 的信息和 hpcusers 组的成员关系。

4.3.2 认证

```
# 正确密码
ldapwhoami -x -D "uid=testuser,ou=people,dc=hpclab,dc=local" -w "Test@1234"
# → dn:uid=testuser,ou=people,dc=hpclab,dc=local  (成功)

# 错误密码
ldapwhoami -x -D "uid=testuser,ou=people,dc=hpclab,dc=local" -w "wrong"
# → ldap_bind: Invalid credentials (49)  (失败)
```

4.3.3 跨节点连通

从 node02 测试连 node01 的 389 端口：

```
python3 -c "
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(3)
s.connect(('10.211.55.11', 389))
print('LDAP port 389: OPEN')
"
# → LDAP port 389: OPEN
```

node02 能连上，说明 LDAP 服务对集群其他节点可用。

服务状态：

```
slapd.service: active (running), enabled
监听：0.0.0.0:389
内存：~3MB
```

4.4 在集群中的作用

LDAP 在集群里做的是统一身份认证。用户登录时，SSH 的 PAM 模块会去 LDAP 查这个用户存不存在、密码对不对。这样管理员只需要在 LDAP 里建一次用户，集群所有节点都能用这个账号登录，不用每台机器都 useradd 一遍。

不过目前只是把 LDAP 服务跑起来了，*还没有配置 PAM 让 node02 实际通过 LDAP 登录*（需要装 `libnss-ldapd` 和 `nslcd`）。

打算继续配置：

Img37请忽略，配置过程有点混乱了

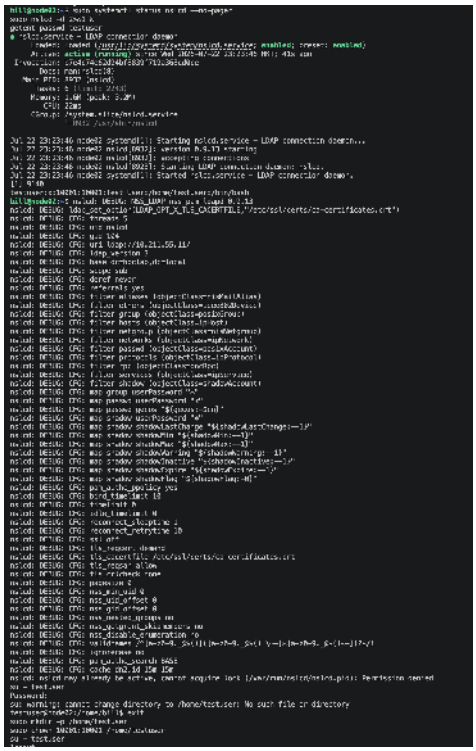

Image 37

Image 38

Agent：对，**在 node02 上用 testuser 登录成功就是成功了**。证明整个认证链路通了：

```
node02 登录 testuser
  → PAM 查本地 /etc/passwd，没有
  → 转给 nslcd
  → nslcd 连 node01 的 LDAP (10.211.55.11:389)
  → LDAP 验证密码正确
  → 登录成功
```

节点	/home/testuser	说明
node01	不存在	没建过
node02	存在	刚才 sudo mkdir 建的

Table 2

LDAP 只管**账号和密码**，不管家目录。家目录是每台机器本地的，需要手动建（或者配 NFS 共享家目录，让所有节点用同一个，那就是存储服务的任务了）。